ECE 575 Final Project: Implementing a 5-stage processor pipeline in Python

Robert Werthman
ECE 575 Section 02
University of Michigan Dearborn

***Abstract*—This document describes an implementation of a 5-stage instruction processing pipeline in the programming language Python. The processor, pipeline stages, instructions, and data memory can be represented in Python using object-oriented programming and python structures like lists and dictionaries. The pipeline is based on the MIPS pipeline describe in [1, p. 301]**

I. Introduction

There are multiple ways to execute instructions in a processor. One way is to execute one instruction in one clock cycle. This is known as single cycle execution. In single cycle execution the processor datapath, all the elements necessary to implement a set of instructions, is designed such that all instructions that are part of the instruction set architecture are executed in a single clock cycle [2, p. 1]. An often used analogy to the execution of an instruction is doing laundry as mentioned in [1, p. 272]. Let's say the instruction you wanted to execute was to wash dirty cloths and put them away. In a single cycle datapath the steps would be

1) Put dirty cloths in washer.
2) Once washer is done put dirty cloths in dryer.
3) When dryer is done put clothes away.

As you can see each step depends on the completion of the previous step. Imagine you had a roommate who was doing laundry after you. The roommate would have to wait for your load of laundry to be done before starting their load of laundry. If you and your roommate did laundry according to single cycle datapath, you would do your load of laundry in the first cycle and your roommate would do their laundry in the second cycle. As it turns out this is not a very efficient to execute instructions–or do laundry. A better way is called the pipelined datapath.

An instruction is a series of steps. All steps need to be completed in order for the instruction to be completed. In a pipelined datapath one step of an instruction is executed in a clock cycle. This is contrary to the single cycle datapath where all of the steps of an instruction are executed in a clock cycle. Using the clothes washing analogy again, the steps in a pipelined datapath would be

1) Put dirty cloths in washer.
2) Once washer is done put dirty cloths in dryer. Roommate puts dirty clothes in washer.
3) Once dryer is done put clothes away. Roommate puts clothes from washer into dryer.
4) Roommate puts clothes away.

You may have noticed the roommate does not have to wait for your clothes to be completely done before starting their laundry. The roommate can use the washer as long as you don't have any clothes in it. This ends up being a more efficient design for a processor datapath than the single cycle. As noted in [1, pp. 286] the steps of an instruction in a pipelined datapath are broken up into 5-stages

1) IF: Instruction fetch
2) ID: Instruction decode and register file read
3) EX: Execution or address calculation
4) MEM: Data memory access
5) WB: Write Back

These stages can be modelled in Python as objects connected to each other. The rest of this document describes the implementation of this model.

II. Architecture

The Python implementation is made of the 5 pipeline stages mentioned in the introduction. These are represented as Python classes. Each of these stages consist of additional components like

- instruction memory
- data memory
- a register file
- an ALU (arithmetic logic unit)
- a PC (program counter)
- a control unit

All of these components belong to different stages of the pipeline except for the control unit which is used in almost all stages of the pipeline. A class diagram of the architecture can be seen in figure 1.

The current architecture only supports a limited number of r-type and i-type instructions. Branch instructions are not currently supported.

A. IF: Instruction Fetch

The instruction fetch part of the pipeline takes the program counter (PC) and fetches the instruction at the address of the program counter [1, pp. 300]. It increments the program counter to the next address. These things are done in a class method called execute which all stages of the pipeline have. Figure 2 depicts the components of the instruction fetch stage.

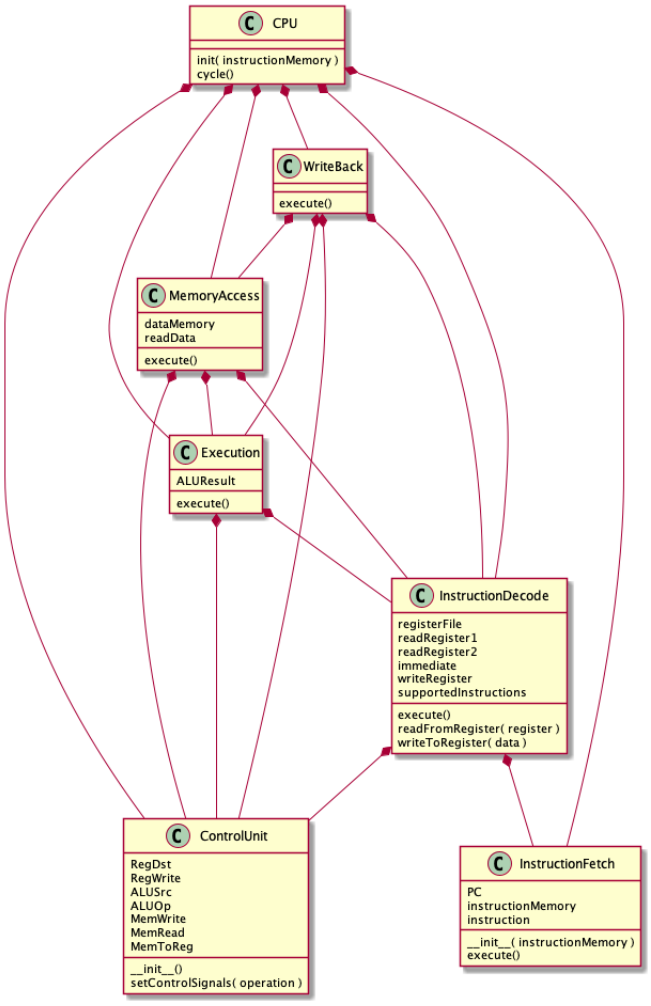

Fig. 1. Class Diagram of Python Implementation

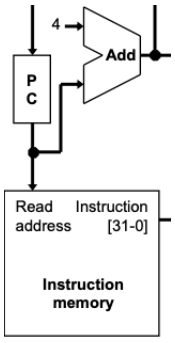

Fig. 2. Instruction Fetch Stage [2]

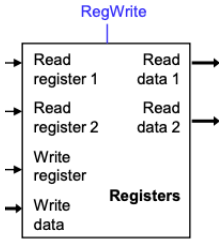

Fig. 3. Instruction Decode Stage [2]

1) Instruction Memory: Instruction memory is represented as a Python list that is passed into the instruction fetch class at initialization. Each instruction itself is also a Python list. Depending on the instruction type the list is made up of the opcode, source and destination registers, address/immediate value, etc. An example of an add instruction in python

```
# An add instruction
instruction = [ "add", "r1", "r2", "r3" ]
```

which means [3, pp. 194]

op	rs	rt	rd

To access the instruction memory you use an index into the instruction memory list. For example

```
# Get the first instruction from instruction memory
instruction = instructionMemory[0]
```

2) Program Counter: The program counter is a python variable that is initialized as 0 and is increment by 1 for every call to the execute method of the instruction fetch class. The program counter cannot exceed the size of the instruction memory list.

B. ID: Instruction Decode

The instruction decode stage gets the instruction from the instruction fetch stage through a class instance member variable. The control signals for the instruction are set in the control unit based on the opcode in the instruction. The read registers, rs and rt, are stored in the instruction decode class as well as the write register, and the immediate/address value if one exists. These come from the instruction retrieved in the instruction fetch stage. Only add, load word (lw), and store word (sw) are supported instructions at this time. The supported instructions are mapped to their instruction type in a dictionary.

```
self.supportedInstrutions = {
            "add" : "rtype",
            "lw"  : "itype",
            "sw"  : "itype"
        }
```

The instruction type is used to determine if there will be an immediate/address in the instruction or not. There are 2 methods that are a part of the instruction decode stage. They are the readFromRegister and writeToRegister methods

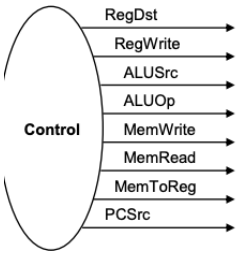

Fig. 4. Datapath Control Unit

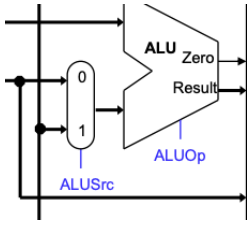

Fig. 5. Execution Stage

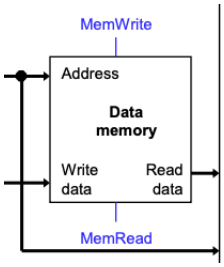

Fig. 6. Memory Access Stage [2]

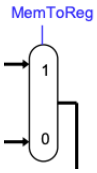

Fig. 7. Write Back Stage [2]

similiar to the read data and write data in [1, pp. 301]. The instruction decode stage has a register file. The components of the instruction decode phase can be seen in 3.

1) Register File: The register file is a dictionary with the register as the key and value of the register as the value.

```
self.registerFile = {
            "r1" : 2,
            "r2" : 1,
            "r3" : 0
        }
```

Only 3 registers are supported at this time.

2) Control Unit: The control unit is a class that contains just member variables which are the control signals used in the different stages of the pipeline. The signals are set to their proper values depending on what the opcode of the instruction is. The control unit is seen in figure 4

C. EX: Execution

In the execution state of the pipeline we compute the result of an arithmetic operation between 2 registers, or compute an address to load or store a value in data memory. The execution stage depends on control signal ALUOp to tell it what type of arithmetic operation to perform. It also depends on ALUSrc to tell it what should be the source of the second ALU operand. The execution stage is seen in figure 5 Right now only the add ALU operation is supported. This allows for an add r-type instruction or the load word or store word i-type instructions.

The result of the arithmetic operation is stored as a class instance member variable that both the Memory Access and Write Back stages will use.

D. MEM: Memory Access

In the memory access stage you can read or write data to/from data memory. Depending on the instruction you may not use the memory access stage at all like in the case of an r-type instruction. This stage is comprised of the data memory and the ability to use an address to read and write data. The memory access stage is depicted in 6.

1) Data Memory: Much like the instruction memory, data memory is represented as a list of integers. The address into data memory is the index of the item in the list you want to read or write.

```
# Data memory
dataMemory = [1, 2, 3, 4]

# Data from first address in data memory
data = dataMemory[0]
```

The ALUResult that comes from the Execution stage is used as an index into the data memory list. Depending on the values of the control signals MemRead or MemWrite you will either be reading or writing to the data memory. An if/else statement is used with these control signals to determine if instruction is to read or write to data memory.

E. WB: Write Back

The Write Back is the simplest of all of the stages, but it is the stage that depends the most on the previous stages. As depicted in [1, pp. 301] and 7 this stage has a multiplexer (MUX) that is controlled by the MemToReg control signal. The value of this signal depends on the instruction being executed. The MUX is represented by an if/else statement with the MemToReg as the if condition.

```
if controlUnit.MemToReg:
    ID.writeToRegister( MEM.readData )
else:
    ID.writeToRegister( EX.ALUResult )
```

F. CPU

The CPU is the main class that creates and contains all of the pipeline stages. It is the class that initializes the instruction

memory, passes it to the instruction fetch stage, and starts the execution of the pipeline.

The the execution of the stages is done in a method called *cycle*. This cycle method simulates a clock cycle and for each instruction in instruction memory execute the pipeline stages in order.

```
for instruction in instructionMemory:
    IF.execute()
    ID.execute()
    EX.execute()
    MEM.execute()
    WB.execute()
```

Once all of the instructions are executed the program ends.

III. Testing and Verification

Test driven development was used when developing this project. This means tests were written, the tests failed, and then code was written to make the tests pass. For each class, a python file containing containing tests was created. These tests were used to verify the behavior of each of the methods of the class. The main driver of verification was the integration tests for the CPU. In those tests a CPU was created that contained all of the stages of the pipeline and an instruction memory with one instruction. The CPU processed the instruction through all of the pipeline stages. At the end of the test, either data memory or a register was checked to see if it contained the right value in the right place.

```
# $r1 = Memory[$r3 + 2]
instructionMemory = [
    [
        "sw", # op
        "r3", # rs
        "r1", # rt
        2 # address
    ]
]
cpu = CPU( instructionMemory )
cpu.cycle()
self.assertEqual( cpu.MEM.dataMemory[2], 2 )
```

The next step for testing is to have an instruction memory with multiple instructions to further verify the behavior of the code.

IV. Open Issues And Future Work

A. Open Issues

The implementation doesn't currently support all instruction types nor does it support all instructions within the types it does support. It has a very small amount of memory data, instruction data, and a tiny register file. As was mentioned in the testing section, the testing has been limited and should be expanded to cover multi instruction instruction memory.

B. Future Work

The implementation is not exactly a pipelined implementation. Only one instruction is run at a time through all of the stages. This is more like a single cycle datapath. In order for it to be a correct pipeline implementation the values shared between each pipeline stage would need an in and out holding

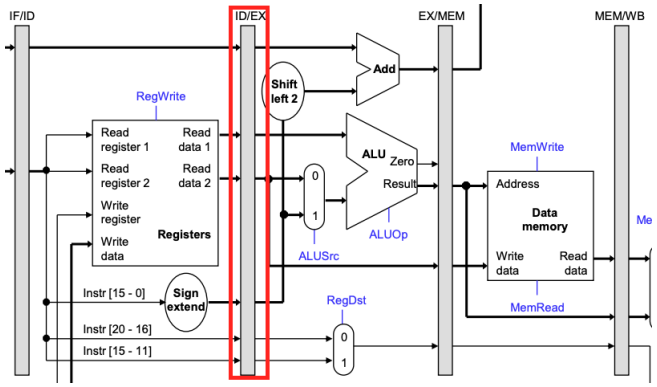

Fig. 8. Pipeline Double Buffer [2]

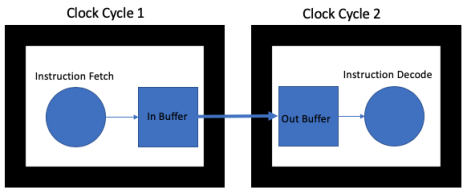

Fig. 9. Double Buffer

area. This area would be similar to a double buffer in computer graphics. One buffer would have the current value being used, the other buffer would contain the value to be used next. If you look at figure 8, the column with the red rectangle represents this idea of a double buffer.

Let's say you have an instruction memory with 2 instructions. For the instruction fetch stage on the first clock cycle, instruction 1 would passed to the **in** buffer of the instruction decode phase. On the next cycle it would be passed to the **out** buffer of the instruction decode phase. Instruction 2 would also be put in the **in** buffer of the instruction decode phase. Once something is in the **out** buffer it can be used in that stage. This means in the second cycle, instruction 1 is put into the out buffer and instruction decode stage is run on this instruction. This is depicted in figure 9.

The source code for this project can be found at https://github.com/rwerthman/ece575

References

[1] J. Hennessy and D. Patterson, *Computer Organization And Design: The Hardware/Software Interface*, 5th ed. Waltham, MA: Elsevier, 2014.

[2] L. Ceze, "A single-cycle MIPS processor". University of Washington, Computer Science and Engineering, CSE 378, Winter 2009. [Online]. Available: https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec07.pdf. [Accessed December 03, 2019]

[3] P. Gillard, "The MIPS instruction set architecture". Memorial University, Department of Computer Science, CS 3725. [Online]. Available: http://web.cs.mun.ca/~paul/cs3725/material/review.pdf. [Accessed December 03, 2019]